Lecture 14: Move Semantics

CS 106L, Fall '20

Last time...

- **Special Member Functions** get called for specific tasks
  - **Copy constructor:** create a new object as a **copy** of an existing object
    `Type::Type(const Type& other)`
  - **Copy assignment:** reassign a new object to be a **copy** of an existing object
    `Type::operator=(const Type& other)`
  - **Destructor:** deallocate the memory of an existing object
    `Type::~Type()`

- SMFs are **automatically** generated for you
  - But if you’re managing pointers to allocated to memory, do it yourself

Custom copy operator

We have to manually copy the elements themselves, not the pointer to array

```
template <typename T>
vector::vector<T>(const vector<T>& other) :
    _size(other._size),                         // copy
    _capacity(other._capacity),                 // copy
    _elems(new T[other._capacity]) {            // newly allocated
    std::copy(other._elems,
                other._elems + other._size, _elems);
}
```

Custom copy assignment

Fill in the blanks! What do we need to copy?

```
template <typename T>
vector& vector::operator=(const vector<T>& other) {
    // fill in the blanks
    // we need to copy other._size, other._capacity, and other._elems
    return *this;
}
```

Custom copy assignment

Fill in the blanks! What do we need to copy?

```
template <typename T>
vector& vector::operator=(const vector<T>& other) {
    _size = other._size;                    // we can copy a size_t
    _capacity = other._capacity;            // we can copy a size_t
    resize(other._size);                    // make sure we have enough space
    std::copy(other._elems,
              other._elems + other._size, _elems);
    return *this;
}
```

Answer in the chat

Which special member functions get called at each stage?

```
vector<string> findAllWords(int i);
int main() {
    vector<string> words1 = findAllWords(12345);  // (1)

    vector<string> words2;                        // (2)

    words2 = findAllWords(23456);                 // (3)
}
```

Answer in the chat

Which special member functions get called at each stage?

```
int main() {        copy constructor
    vector<string> words1 = findAllWords(12345);   // (1)

                                destructor
    vector<string> words2;                         // (2)
copy assignment
    words2 = findAllWords(23456);                  // (3)
}
                  destructor
```

Answer in the chat

Copy elision allows us to optimize out the first set of calls…

~~copy constructor~~

```
int main() {
    vector<string> words1 = findAllWords(12345);   // (1)
```

~~destructor~~

```
    vector<string> words2;                         // (2)
```

copy assignment

```
    words2 = findAllWords(23456);                  // (3)
}
```

destructor

😢 but we still can't optimize out making two **vectors** on line 3.

for now, let’s ignore the effects of copy elision

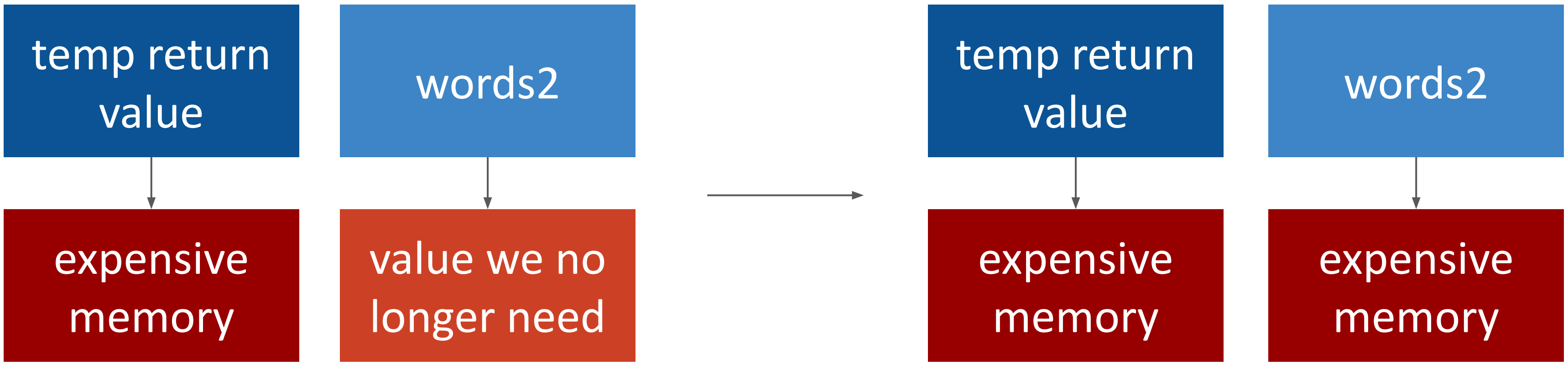
The central problem

```
words2 = findAllWords(23456);
```

We need some way to **move** the resources of the result of **findAllWords** to words2, so that we don't create two objects (and immediately destroy one).

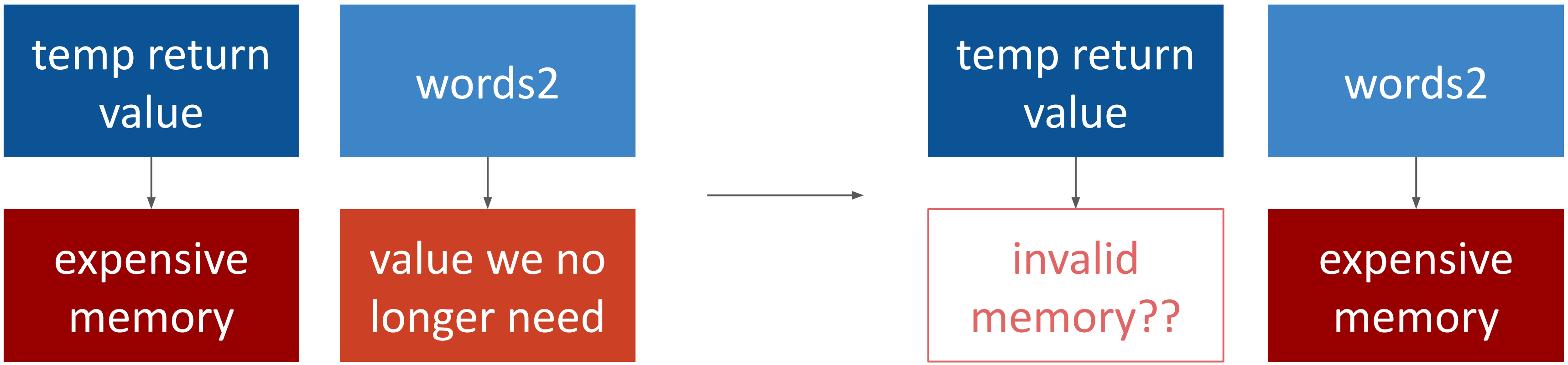
(Question in the chat: why don't we return **vector&** instead of **vector**?)

Copy



Why are we copying something we no longer need?

Moves



This works because we **no longer need** the temp value

One idea

What if **= always represented a move?** Does this work? (answer in chat)

move constructor

```
int main() {
    vector<string> words1 = findAllWords(12345);

    vector<string> words2;
```

move assignment

```
    words2 = findAllWords(23456);
}
```

✓ **Yes, this works correctly!** It is also efficient (no wasted memory).

Another example

What if **= always represented a move?** Does this work? (answer in chat)

move constructor

move assignment

```
int main() {
    vector<string> words1 = findAllWords(12345);

    vector<string> words2 = words1;

    words1.push_back(“Everything is fine!”);
}
```

🚫 **No, this doesn't work! words2** stole its array from **words1** :(

Another example

To get this to work correctly, we need to use a combination…

move constructor

copy assignment

```
int main() {
    vector<string> words1 = findAllWords(12345);

    vector<string> words2 = words1;

    words1.push_back("Everything is fine!");
}
```

How can we distinguish between these two cases?

Topics

- lvalues vs. rvalues
- Move implementation
- Forcing a move to occur
- swap and insert

lvalues vs. rvalues

Looking more closely...

Recall: we could use **move** on the first line because it was **temporary**.

```
int main() {
                        move constructor
    vector<string> words1 = findAllWords(12345);
                        copy assignment
    vector<string> words2 = words1;
}
```

L-values and r-values generalize the idea of "temporariness."
A **r-value** is "temporary," and a **l-value** is not.

Official definition: a l-value has an address (can do &), and a r-value does not

A l-value can appear left or right of =.

```
x = 5;
y = x;
```

An r-value can only appear right of =.

~~5 = x;~~

```
y = 5;
```

Which of these are r-values? (Answer in chat)

```
int val = 2;                           // A
int* ptr = 0x02248837;                 // B
vector<int> v1{1, 2, 3};               // C
auto v4 = v1 + v2;                     // D
auto v5 = v1 += v4;                    // E
size_t size = v.size();                // F
val = static_cast<int>(size);          // G
v1[1] = 4*i;                           // H
ptr = &val;                            // I
v1[2] = *ptr;                          // J
```

Which of these are r-values? (Answer in chat)

```
int val = 2;                          // A
int* ptr = 0x02248837;                // B
vector<int> v1{1, 2, 3};              // C
auto v4 = v1 + v2;                    // D
auto v5 = v1 += v4;                   // E
size_t size = v.size();               // F
val = static_cast<int>(size);         // G
v1[1] = 4*i;                          // H
ptr = &val;                           // I
v1[2] = *ptr;                         // J
```

A l-value’s lifetime is until end of scope…

A r-value’s lifetime is until end of line

unless you artificially extend it

References

A reference is an alias to an existing object

```
int main() {
   vector<int> vec;
   change(vec);
}

void change(vector<int>& v) {...}   // v is an alias of vec
```

Notice: vec is a **l-value**.

What happens when you try to reference a r-value?

```
int main() {
    change(7);
}

void change(int& a) {...}   // this doesn't work
```

- One caveat: in order to pass a variable by reference, you need to actually have a variable. The following does not work, for our example above:

```
doubleValueWithRef(15); // error! cannot pass a literal value by reference
```

Compiler error:

```
../all-examples.cpp:135:5: error: no matching function for call to 'doubleValueWithRef'
  doubleValueWithRef(15);
  ^~~~~~~~~~~~~~~~~~
../all-examples.cpp:11:6: note: candidate function not viable: expects an l-value for 1st argument
void doubleValueWithRef(int &x);
     ^
1 error generated.
```

l-value reference

```
int main() {
    int i = 7;
    change(i);
}

void change(int& a) {...}
```

r-value reference

```
int main() {
    change(7);
}


void change(int&& a) {...}
```

Why rvalue references?

Copy constructor (**lvalue&** reference)

```
vector<T>(const vector<T>& other) :
     _size(other._size),
     _capacity(other._capacity) {
     // have to manually copy the array
     _elems = new T[other._capacity];
     std::copy(other._elems,
               other.elems_ + other._size,
               _elems);
}
```

Have to leave **other** in an usable state (**const**).

Move constructor (**rvalue&&** reference)

```
vector<T>(vector<T>&& other) :
     _size(other._size),
     _capacity(other._capacity) {
     // steal the other array 🕵️
     _elems = other._elems;

     other._elems = nullptr;
     other._size = 0;
}
```

Brutally rob **other**’s resources, because it is temporary and will disappear when the function exits.

Overloading between & and && versions of the same function allows us to disambiguate between copy and move!

but wait

An issue

Move constructor (rvalue&& reference)

```
vector<T>(vector<T>&& other) :
     _size(other._size),
     _capacity(other._capacity) {
     // steal the other array 🕵️
     _elems = other._elems;

     other._elems = nullptr;
     other._size = 0;
}
```

this is a lvalue
(because **`other&&`**, a reference, is a lvalue)

so this performs… [answer in the chat]
a copy :(
Copying a pointer is cheap, but you could imagine situations where this is a problem

Fixing this

Move constructor (rvalue&& reference)

```
vector<T>(vector<T>&& other) :
     _size(std::move(other._size)),
     _capacity(std::move(other._capacity)) {
     // steal the other array 🕵️
     _elems = std::move(other._elems);

     other._elems = nullptr;
     other._size = 0;
}
```

std::move is a cast to a rvalue&&
(equivalent to std::static_cast<T&&>)

Move Assignment

Move assignment (rvalue&& reference)

```
vector<T>& operator=(vector<T>&& other) {
     _size = std::move(other._size);
     _elems = std::move(other._elems);
     _capacity = std::move(other._capacity);
     other._elems = nullptr;
     other._size = 0;
}

vector<T>& v = {1, 2, 3};
vector<T>& v2;
v2 = std::move(v);
```

std::move is a cast to a rvalue&&
(equivalent to std::static_cast<T&&>)

Takeaways

- Use a **constructor** taking a **rvalue** for move constructor
- Use **operator=** taking a **rvalue** for move assignment
- Use **std::move** to make sure other object's values are treated as rvalues (and so moved)
    - Call **std::move** to force anything to become a rvalue (and get its data taken!)

std::move does not move anything

Recall...

A l-value's lifetime is until end of scope...

A r-value's lifetime is until end of line

unless you artificially extend it

For our vector...

```
template <typename T>
void vector<T>::push_back(const T& element) {
    elems[_size++] = element;           // equals → copy
}

template <typename T>
void vector<T>::push_back(T&& element) {
    elems[_size++] = std::move(element);  // move!
}
```

std::swap

```
template <typename T>
void swap(T& a, T& b) noexcept {
    T c(std::move(a)); // move constructor
    a = std::move(b);  // move assignment
    b = std::move(c);  // move assignment
}
```